Solving the Class Responsibility Assignment using Java and ATL

Leif Arne Johnsen
h138166@stud.hib.no

Fernando Macías
fmac@hib.no

Adrian Rutle
aru@hib.no

Bergen University College
Norway

Abstract

The objective of Class Responsibility Assignment (CRA) is to produce high-quality designs for object-oriented systems. It deals with the problem of finding a good class diagram for a given set of methods and attributes with functional and data relationships among them [?]. Solving this problem is not a trivial task and often involves manual intervention. In this paper we describe an automatic solution[1][2] based on the simulated annealing algorithm. We present also an evaluation of the solution's performance w.r.t. execution time and CRA-index.

1 Introduction

To solve the Class Responsibility Assignment (CRA) problem we use Java to evaluate models and to keep track of the best solution generated. We use ATL to generate new solutions. ATL is called programmatically from eclipse. We use local search to find a suitable output model. Simulated annealing is implemented to avoid getting stuck in a local optima, in the beginning of the search. A local optima is when you can make any change to the current solution, but it will never make it better, improving the solution requires making changes in more than one step. Simulated annealing may accept slightly worse solutions in the beginning stages of the algorithm, but gets stricter at each iteration. This allows it to escape a local optima

2 Solution description

2.1 Rules

We have created four main rules in order to create classes, assign features, reassign features and delete empty classes. The first rule creates an empty class for each feature in the class model. It also sets the name of the classes created, each class is named from 1 to the number of features in the class model. We start with an index which is initialized to 0 and incremented every time a new class is created to keep track of the classes created. The second rule randomly assigns each feature to a class, but the empty classes are still kept. The third rule randomly reassigns features. It goes through each feature. When matched with a feature, it will randomly generate a number between 0 and 1. If the random number is less than 1 divided by the number of features then the feature is reassigned to a random class. The fourth rule deletes all empty classes.

The first and second rules could perhaps be combined into one. We had some troubles doing this with ATL because classes were created at each feature match, but at each feature (except the last) only a subset of the

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

[1]`https://bitbucket.org/leifarnemsc/solutionone`

[2]`http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_JavaAtl.vdi`

classes were created. Therefore the classes created early would gain an unfair advantage if you randomly assigned a feature to the classes created so far.

The third rule is the only one that is called multiple times, it is called on every iteration of the search. It is used to slightly modify the solution. The fourth rule is perhaps not necessary, especially w.r.t. the constraint which is specified in the metamodel.

In addition to these four rules, we have used some other rules. These rules were needed to copy the information from the input model to the output model since we have implemented an in-place transformation. The copy rules could have been avoided if we had used refining mode.

2.2 Simulated annealing

The algorithm gets its name from the idea of forging iron. Starting at a high temperature, the iron is easy to bend. The temperature is decreased with time, as the iron gets colder it gets harder to bend.

2.3 The algorithm

1. First generate a random class diagram
2. Calculate its CRA-Index
3. Generate a random neighbouring class diagram
4. Calculate the CRA-Index of the new class diagram
5. Compare them, if the new CRA-Index is better then move to the new class diagram, else maybe move to the new class diagram
6. Reduce temperature
7. Repeat steps 3 to 6 until the temperature is below the minimum temperature

2.4 Each step in-depth

First generate a random class diagram. This is done using the first and second rule, described earlier. After generating a class diagram, set it as best.

Calculate its CRI. To calculate the CRA-Index of the generated class diagrams, the provided CRAIndex-Calculator is used. The craBest is set at this point, which will refer to the value of the currently best class diagram.

Generate a random neighbouring class diagram. New class diagrams are generated from the currently best class diagram, it is done by applying the third rule (re-assign) on the currently best diagram. We take the output from that transformation and set it as the new class diagram. **Calculate the CRA-Index of the new class diagram.** Same as before, but only now you give the new class diagram as input to the evaluator function. Set craNew to the value returned from the evaluation.

Compare them. When you compare them you get two possibilities, if the new class diagram is better then you accept it and move on. However if the new class diagram is worse or has the same value then you might still accept it. The reason for accepting worse class diagrams is that we want to avoid getting stuck in a local optima. Although we can not accept a worse class diagram every time. Instead we want to take a calculated risk, which means that the bigger difference in CRA-Index between the best and new class diagram, the less likely the new class diagram is to get accepted. We call the likelihood of accepting a new class diagram for acceptance probability.

Acceptance probability We introduce a function to calculate the acceptance probability, it takes as input the CRA-Index of the best class diagram, the new class diagram and the current temperature of the system. If the craNew > craBest then return 1.0 else return

$$e^{\frac{craNew-craBest}{T}}$$

Reduce temperature. After each iteration the temperature is decreased geometrically, that is multiplying it by a factor < 1. This factor is calculated based on the initial temperature, the ending temperature, and the amount of iterations you wish to loop through. It returns the result of this calculation $(\frac{T_{min}}{T_{max}})^{\frac{1}{n}}$. There is not anything special about this function, it only tells you which factor to use in order to get from the starting temperature to the ending temperature in n iterations.

2.5 Choosing the parameters to use for the search

The algorithm is easy to implement, the challenge lies more in giving the correct parameters, meaning choosing the right starting temperature, deciding how fast the temperature should decrease and lastly choosing which temperature to end the search at.

We have used a temperature of 3 and an ending temperature of 0.01. The cooling schedule is derived from the starting temperature, the ending temperature and the number of times we want to iterate. Formula used for the schedule is $(\frac{T_{min}}{T_{max}})^{\frac{1}{n}}$ where n is the number of iterations, T_{min} is the end temperature and Tmax is the starting temperature.

The iterations is set manually depending on the input model to evaluate, however we think that these parameters should have been derived from the input models. Finding these relations however requires analysis of several output models created from the same input model, using different parameters.

2.6 Program flow code

```
//Create a class for each feature
transform(metaModel, inModel, tempModel, TRANSFORMATION_DIR, "Model2Classes");
//Generate a random solution
transform(metaModel, tempModel, bestModel, TRANSFORMATION_DIR, "reassignAll");

double temperature = 3;
double endTemperature = 0.01;
double k = calculateCoolingFactor(endTemperature, temperature, 1875);
//Evaluate the generated solution
double craBest = evaluate(bestModel);

while(temperature > 0.01){
        tempModel = EmftvmFactory.eINSTANCE.createModel();
        tempModel.setResource(rs.createResource(URI.createURI("src/temp.xmi")));
        //Generate a new solution from the old solution
        transform(metaModel, bestModel, tempModel,
                TRANSFORMATION_DIR, "ReassignFeatures");

        //Evaluate the new solution
        double craNew = evaluate(tempModel);

        //Compare
        if(acceptanceProbability(craBest, craNew, temperature) > r.nextDouble()){
                //update best solution
                bestModel = tempModel;
                craBest = craNew;
        }//end if

        //Decrease temperature
        temperature *= k;
}//end while

//Delete empty classes from the best solution and transfer it to the outmodel
transform(metaModel, bestModel, outModel, TRANSFORMATION_DIR,
"deleteEmptyClasses");

//Save the best solution
try {
        outModel.getResource().save(Collections.emptyMap());
} catch (IOException e) {
        e.printStackTrace();
}
```

2.7 Evaluation

Correctness and completeness criteria were met for all output models, created from the input models. The CRA-Index and the execution time for the output models are given in the table below.

Output from	A	B	C	D	E
CRA-Index	3.0	3.5	-1.25884	-15.72579	-84.61894
Performance	37.533ms	74.847ms	186.919ms	586.493ms	1927.053ms
Iterations	1875	3750	7500	15000	30000

References

[BBL10] Michael Bowman, Lionel C. Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Trans. Software Eng.*, 36(6):817–837, 2010.

[Fle16] Martin Fleck. Solution using MOMoT. `https://github.com/martin-fleck/cra-ttc2016/blob/master/MOMoT_solution/TTC2016_CRA_MOMoT.pdf`, 2016.

[Fou16] The Eclipse Foundation. ATL User Guide. `http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`, 2016.

[Gel16] Katrina Ellison Geltman. Simmulated Annealing. `http://katrinaeg.com/simulated-annealing.html`, 2016.

[Joh16] Leif Arne R. Johnsen. Online demo: XP-TUe_JavaAtl.vdi. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_JavaAtl.vdi`, 2016.

[Wag16] Dennis Wagelaar. EMFTVM Online Documentation. `https://wiki.eclipse.org/ATL/EMFTVM`, 2016.